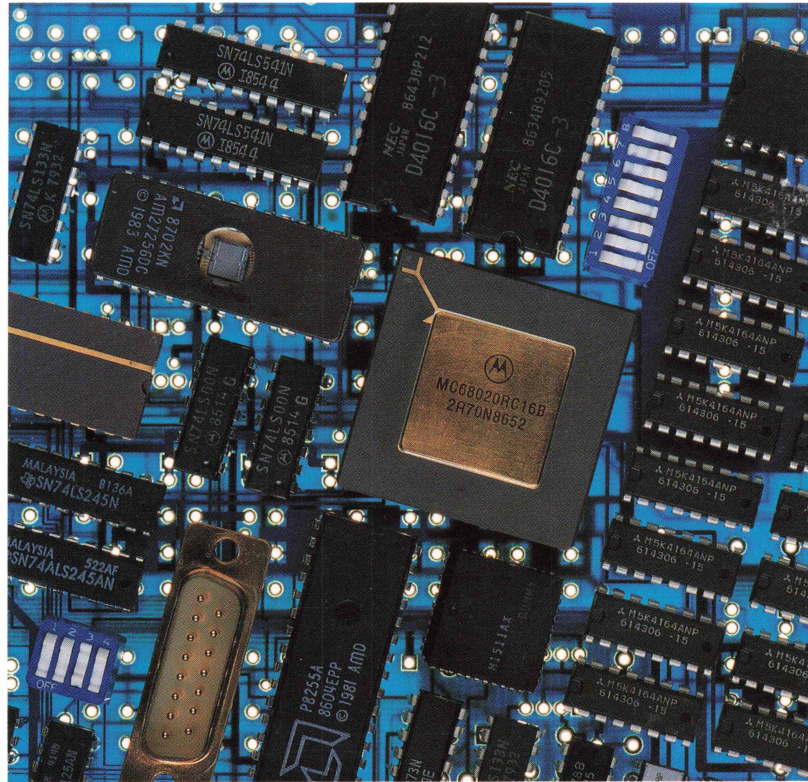


MICROPROCESSOR • SOLUTIONS

PRINCIPLES OF MICROPROCESSOR SYSTEMS



Introduction

FLUKE

M I C R O P R O C E S S O R • S O L U T I O N S

P R I N C I P L E S O F M I C R O P R O C E S S O R S Y S T E M S

FLUKE

©1988, John Fluke Mfg. Co., Inc. All rights reserved. Litho in U.S.A.

John Fluke Mfg. Co. Inc.

Customer Support Services

P.O. Box C9090 MS 239D

Everett, Washington 98206

(206) 347 6100

Table of Contents

Section 1: Introduction

Microprocessor Functions 1-2

A Sample Microprocessor-based System 1-3

Input/Output (I/O) 1-4

Decision-Making 1-5

Storage 1-5

Section 2: Numbering Systems

Numbering Systems 2-1

Section 3: Digital Logic

Logic Gates 3-1

AND Gate 3-1

OR Gate 3-2

Inverter or NOT Gate 3-2

Nand Gate 3-3

NOR Gate 3-3

Negative-AND and Negative-OR Gates 3-4

Tri-state Buffer 3-5

Section 4: The Micro-Computer

The Microprocessor 4-2

The Microprocessor Bus 4-3

The Address Bus 4-3

Data Bus 4-7

Section 5: Memory

Semiconductor Memory	5-1
Read-Only Memory (ROM)	5-1
ROM Data Organization	5-4
ROM Usage	5-5
Random Access Memory (RAM)	5-6
	5-10

Section 6: Input/Output

What is an I/O device?	6-1
I/O Addressing	6-2
I/O Interfaces	6-4
Bus-width Considerations	6-6

Section 7: Interrupts & Direct Memory Access

Interrupts	7-1
Types of Interrupts	7-3
Hardware Interrupts	7-3
Software Interrupts	7-4
Exceptions	7-5
Direct Memory Access	7-6
The DMA Controller	7-6
Types of DMA Transfers	7-7
Typical Uses for DMA	7-7

Section 8: Summary

Section 1

Introduction

This book was written to explain how microprocessor-based systems work. Prior knowledge of computers or microprocessors is helpful, though not necessary, in understanding the materials covered in this course.

Topical Outline:

- I. Introduction: Microprocessor Functions
 - A. Input/Output
 - B. Decision-making
 - C. Storage
- II. Numbering Systems
- III. Digital Logic
 - A. Logic Gates
- IV. The Micro-Computer
 - A. Microprocessor
 - B. The Microprocessor Bus
- V. Memory
 - A. Read-Only Memory
 - B. Random Access Memory
- VI. Input/Output
 - A. Memory Mapped Addressing
 - B. I/O Mapped Addressing
 - C. I/O Interfaces
- VII. Interrupts & Direct Memory Access
 - A. Hardware Interrupts
 - B. Software Interrupts
 - C. Exception Interrupts
 - D. Direct Memory Access

Microprocessor Functions

What is a microprocessor-based system? Obviously, it is a system containing a microprocessor, but how is it different from other systems not incorporating a microprocessor?

Most electronic devices today contain microprocessors, or μ Ps, for short. Computers, of course, are microprocessor-based, but other common household and consumer items contain microprocessors as well, such as programmable thermostats, microwave ovens, electronic ignition systems in automobiles, and childrens' toys, to name a few.

A Sample Microprocessor-based System

Let's look at the programmable thermostat as an example. These household items usually feature a digital clock and various buttons used in setting on and off times for the heating and ventilation system throughout the week. There are usually no moving parts other than an electronic relay to turn the furnace or air conditioner on or off and are *programmed* electronically through the control panel.

Conventional thermostats contain many moving parts, and rely on an electric motor to turn a wheel, which in turn trips various switches. They also rely on gravity, bi-metallic strips, and mercury switches to sense and react to temperature variations. These devices are *set* by adjusting levers and mechanical switches and can generally only be configured for one 24-hour cycle.

A closer look at our electronic thermostat reveals the basic building blocks of a microprocessor-based system:

1. **Input** – control panel buttons, temperature sensor
2. **Output** – digital display, clock/calendar, control line to furnace
3. **Decision-Making** – Microprocessor or Central Processing Unit (CPU)
4. **Storage** – embedded program memory, user input memory (for programmed settings)

All microprocessor-based systems have these four elements in common. Larger systems, such as computers, just have more complex versions of these same four components.

Input/Output (I/O)

A system is meaningless unless it has the capability to communicate with the outside world. This communication is generally referred to as *Input/Output*, or *I/O*.

In the most general sense, the purpose of the microprocessor is to accept as input data from the world outside of itself, manipulate that data in some predefined way, and provide some form of resultant output to the outside world. In the case of our electronic thermostat, the input would come from two sources: (1) the person pressing the control panel buttons and (2) the temperature sensor. The buttons translate the person's programming keystrokes into signals which the microprocessor then uses in its own embedded program. Likewise, the temperature sensor continually feeds temperature data to the microprocessor to use in its program.

Based on the input from the control panel and the temperature sensor, the thermostat's resultant output is a signal to the building's heating system to either switch on or off (or stay switched on or off).

Decision-Making

The signal sent to the furnace depends on the ambient temperature of the room. It also depends on the programmed settings input by the operator and the program embedded within the thermostat. The *Central Processing Unit (CPU)* performs the decision-making required by the system. Collectively, the components of the CPU and additional specialized circuits make up the microprocessor in one small package.

It is the CPU's job to take the data previously sensed from the control panel and the continuously sensed data from the temperature sensor and, according to an embedded set of instructions in memory (a program), decide which signal to assert on the output line going to the furnace. This program instructs the CPU to take into account the time of day, day of week, high/low settings input by the operator, present ambient temperature, and a multitude of other information in its decision-making process. Exactly how the CPU accomplishes this task will be covered later.

Storage

The embedded program, as well as the programmed settings input by the operator are stored for subsequent retrieval in *memory*. Memory is made up of individual *cells*, each capable of holding one *bit* of data represented by either a "1" or "0". By arranging these cells in *bytes* (groups of 8 bits), coded binary data (data consisting of "1"s and "0"s) can be represented. The microprocessor is capable of reading and interpreting these data patterns from memory as well as writing data patterns to memory.

In our electronic thermostat we have described how a previously mechanical device could be made more versatile by incorporating the microprocessor. Through it, we have described the basic four elements of a microprocessor-based system.

Section 2

Numbering Systems

To understand how computers and other digital systems work, an understanding of their numbering system is necessary. This chapter covers the fundamental numbering systems incorporated in today's microprocessor-based systems.

Numbering Systems

Because humans have two hands, each with four fingers and a thumb, it is natural for us to count by fives and tens. If we had ten of something, each item could correspond to one of our ten digits. Long ago, abstract symbols or pictograms were scrawled on tablets or stone to record these quantities. This is how our ancestors began counting, and when large numbers of things exceeded the number of symbols we had or digits on our hands and feet, stones or pebbles stored in a pouch more than likely presented a suitable solution.

The need to count larger quantities of things forced us to create abstract systems of numerals. We began to count in *groups*, and invented additional symbols to represent these groups. A familiar example fashioned after pictograms of architectural columns, was loosely based on counting in groups of five—namely, the Roman Numeral system. The numerals in common use today are Arabic Numerals, utilizing the *decimal system*, based on groups of ten.

Decimal Numbers

In our Arabic number system, numbers are expressed in groups of ten, or *base-10*, due possibly to the fact that we have ten fingers. We have ten numerals, 0 through 9, to represent our *decimal* numbering system. In use, these numerals are placed side-by-side with each numeral representing a *digit* (the Latin word for “finger” or “toe”), of a number. Each digit in our numbering system is *weighted*—representing an increasing order of magnitude based on 10, see Figure 2-1. Counting from right to left, the first digit represents the *ones*, or $n \times 10^0$; the next digit representing $n \times 10^1$, or how many *tens* are in the number; the third digit $n \times 10^2$, or how many *hundreds*, and so forth.

The number 236_{10} , therefore, is the sum of 2×10^2 , 3×10^1 and 6×10^0 , or $200 + 30 + 6$. (The subscripted “10” indicates that the number “236” is base-10.)

10^2 Column	10^1 Column	10^0 Column
2	3	6
2×10^2 200	3×10^1 30	6×10^0 6

Figure 2-1 Column Weights

Binary Numbers

Microprocessors, on the other hand, cannot use our decimal system very efficiently. Since they are digital electronic devices they require a binary system (base-2) comprised of the two numerals zero and one which can be represented and manipulated by electronic circuits. In a binary system, our decimal number 236_{10} winds up being 11101100_2 . This number can be easily stored by eight electronic *latches* collectively comprising one of the microprocessor’s registers, for example. As in our decimal number, each digit in this binary number is weighted, except they are weighted by a power of *two* instead of ten.

Let's examine the number 11101100_2 for a moment. If each digit is *weighted* by a power of two, then each digit signifies the presence ("1") or absence ("0") of a given quantity, and that quantity is dependent on the digit's *weight*. See Figure 2-2.

Column Weight							
2^7 Column	2^6 Column	2^5 Column	2^4 Column	2^3 Column	2^2 Column	2^1 Column	2^0 Column
1	1	1	0	1	1	0	0
1×2^7 128	1×2^6 64	1×2^5 32	0×2^4 0	1×2^3 8	1×2^2 4	0×2^1 0	0×2^0 0

Figure 2-2 *Binary Column Weights*

The number 11101100_2 therefore, is the sum of 1×2^7 , 1×2^6 , 1×2^5 , 0×2^4 , 1×2^3 , 1×2^2 , 0×2^1 , and 0×2^0 , or $128 + 64 + 32 + 0 + 8 + 4 + 0 + 0$, or 236_{10} .

Decimal to Binary Conversion

Unfortunately, it's rather difficult to quickly think and write in binary: the decimal number $67,217_{10}$, for instance, is 10000011010010001_2 , so conversion is generally necessary. To convert a decimal number to binary, repeated division by two is performed: $67,217_{10}$ is divided by two with each resulting quotient again divided by two. It is the sequence of remainders that form the actual binary number. The first remainder is placed in the 2^0 column, the second in the 2^1 column, and so on.

67,217 / 2 =	33,608	remainder 1
33,608 / 2 =	16,804	remainder 0
16,804 / 2 =	8,402	remainder 0
8,402 / 2 =	4,201	remainder 0
4,201 / 2 =	2,100	remainder 1
2,100 / 2 =	1,050	remainder 0
1,050 / 2 =	525	remainder 0
525 / 2 =	262	remainder 1
262 / 2 =	131	remainder 0
131 / 2 =	65	remainder 1
65 / 2 =	32	remainder 1
32 / 2 =	16	remainder 0
16 / 2 =	8	remainder 0
8 / 2 =	4	remainder 0
4 / 2 =	2	remainder 0
2 / 2 =	1	remainder 0
1 / 2 =	0	remainder 1

10000011010010001

The result, then, is 10000011010010001_2 . Larger numbers require even more division. This enormous disparity between decimal and binary numbering systems has led to the development of the *hexadecimal* number system, also called the *hex* number system.

Hexadecimal Numbers

Binary numbers are frequently handled in groups of eight digits called a *byte* with each of the eight digits called a *bit*, for *Binary digIT*. In addition, four bits are sometimes referred to as a *nibble*.

Microprocessors, and computers in general usually deal with binary numbers in groups of four or eight bits. A numbering scheme based on four- or eight-bit values was needed to bridge the gap between binary and decimal systems.

Hexadecimal numbers are based on the number 16 with sixteen digits represented by 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. To understand why base-16 was used, let's compare it with the binary numbers from zero to sixteen:

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

It appears that the first sixteen binary numbers cycle through all the possible zero-one combinations represented by four bits. This means that every four bits can be replaced by one hexadecimal digit—precisely what was needed to simplify dealing with digital data.

To see this correspondence, convert a hex number to binary by simply converting each digit into its binary counterpart:

B5F2₁₆

B	5	F	2
↓	↓	↓	↓
1011	0101	1111	0010

1011010111110010₂

To convert from binary to hex, simply group the binary number by nibble and convert each nibble. If necessary, pad the most significant digits with zeros:

100101110111001110₂

10	0101	1101	1100	1110
0010	0101	1101	1100	1110
↓	↓	↓	↓	↓
2	5	D	C	E

25DCE₁₆

It may take some practice getting used to dealing with hex numbers. Remembering to use “A” instead of “10” and thinking in “sixteens” will get easier as you work through this book.

Remember that counting in hex is really no different than counting in decimal—when you run out of numerals, increment the next higher digit and start counting over:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D . . .

When dealing with a single hex digit, remember: its range is 0 through F (15₁₀); two hex digits range from 0 through FF (255₁₀), and four hex digits from 0 to FFFF (65,535₁₀).

Octal Numbers

Another number system worth mentioning here is the Octal Number system, with eight numerals, developed primarily for dealing with binary numbers in older mainframe computer systems. These numerals are 0, 1, 2, 3, 4, 5, 6, and 7. Like the hexadecimal system, it's an abbreviated way of notating digital data, however, instead of representing four-bits of data per digit, octal numbers represent three-bits of binary data:

7562 ₈			
7	5	6	2
↓	↓	↓	↓
111	101	110	010
111101110010 ₂			

Converting from octal to hexadecimal is a simple matter of first converting the number to binary, then regrouping the bits into nibbles, then converting to hex.

Most of the numbers we will be dealing with will be in decimal or hex. Generally, when dealing with quantities, decimal numbers will be used, and when dealing with memory addresses and data, hex numbers will be used.

Section 3

Digital Logic

Logic gates are the building blocks of digital systems. Combined with the resistor, capacitor, and diode, they are what make up the microprocessor, memory, and circuits in a digital system.

Logic Gates

A logic *gate* is a circuit that performs a certain logic operation on a signal or set of signals present at the circuit. Throughout this book, we will speak in terms of *positive-logic*, that is, a high signal is denoted by a logic “1” and a low signal a logic “0”. In addition, a circuit’s input is considered *active-high* when a high input causes the circuit to turn on, or activate, and is considered *active-low* when a low input causes it to turn on.

AND Gate

We will cover the basic logic gates and their characteristics, starting with the AND gate. It consists of two or more inputs with one output. All inputs must be high (1) for the output to be high. In the two-input AND gate in Figure 3-1 both inputs A and B must be high to turn the gate on (produce a high at output Y). The *truth table* for the gate appears below its symbol. Truth tables are used to illustrate output states for all possible inputs for a given gate or circuit. As can be seen from the table, if *any* input is low (0), the gate is turned off and the resulting output is low.



Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3-1 AND Gate Symbol and Truth Table

OR Gate

The OR gate, like the AND gate, consists of one or more inputs and an output. See the example to the right of a two-input OR gate (Figure 3-2). The output (Y) is high if any of the inputs (A or B) are high, and the output is low only when all inputs are low. Compare the truth table here with the truth table for the AND gate on the previous page. What are the differences?

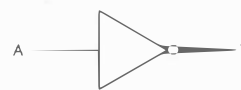


Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 3-2 OR Gate Symbol and Truth Table

Inverter or NOT Gate

Perhaps the simplest gate is the *Inverter*, or *NOT gate*, Figure 3-3. It has one input (A) and one output (Y) with a *negation indicator* (the bubble or circle) on its output. Its purpose is to provide at its output an inverted, or complementary signal to the one applied at its input: if a high is present at the input, a low would be output, conversely, a low input would produce a high output.



Input A	Output Y
0	1
1	0

Figure 3-3 Inverter Symbol and Truth Table

Figure 3-4 shows how an inverter can be placed at the output of a gate to *complement* its output. The inverter is used in conjunction with an AND or OR gate to produce two additional gates: the NOT-AND (NAND) and NOT-OR (NOR) gates, described on the following page.

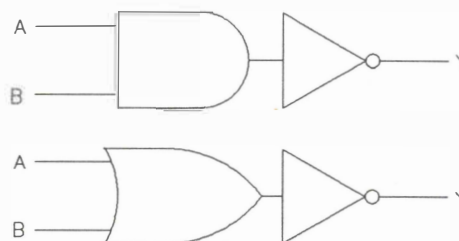
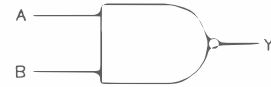


Figure 3-4 Using the Inverter with AND and OR gates

Nand Gate

The NAND-gate is a combination of the inverter and the AND-gate (“NAND” is the contraction of “Not-AND”). It consists of one or more inputs with an inverted output. The symbol and truth table appear in Figure 3-5. The inputs A and B feed what appears to be an AND-gate with an inversion bubble at its output, Y. This gate is logically equivalent to the NOT-AND gate combination described on the previous page.

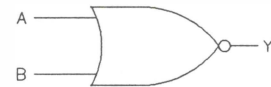


Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Figure 3-5 NAND Gate Symbol and Truth Table

NOR Gate

The NOR gate, Figure 3-6, is a combination of the inverter and OR gate. It is made up of two or more inputs with an inverted output, like in the NAND gate. In this gate, if any inputs are high, the output is low, and the output is high if all inputs are low. This gate is logically equivalent to the NOT-OR gate combination described on the previous page.



Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 3-6 NOR Gate Symbol and Truth Table

Placement of the inversion bubble on a gate’s output is the same as connecting an inverter to it: it negates, or inverts the output to the opposite state. It is likewise possible to place an inversion bubble on a gate’s *input* to invert the signal *entering* the gate. This completely alters the gate’s logic, however, as we will see on the next page.

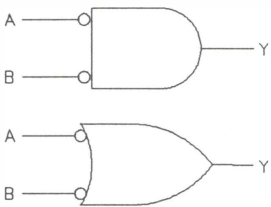
Negative-AND and Negative-OR Gates

AND and OR gates with inverters at their inputs behave in the same manner as the NOR and NAND gates, respectively. Yes, you read correctly: the Negative-AND gate is logically equivalent to the NOR gate, and the Negative-OR gate is equivalent to the NAND gate. The symbols and truth tables for both gates appear in Figure 3-7, and if compared with the truth tables for the NOR and NAND gates, the same data would be found.

Why distinguish between these virtually identical gates? To answer this, examine the truth tables for each of these devices. In the case of the Negative-AND gate, in order to produce a high, all inputs must be low. Logically, it's easier to think of this gate as an AND gate that has *active-low* inputs. In other words, to activate this gate, you need all lows. In the case of the Negative-OR gate, in order for the output to be high, any of the inputs can be low. Again, this gate can be easier understood as an OR gate with *active-low* inputs.

Using their equivalent NOR and NAND gates, the logic is more difficult to follow. These gates have *active-low outputs*, and therefore, are low when the gate is on, or activated. In real applications, it is often more advantageous to use less confusing symbology, as in the case of the Negative-AND and Negative-OR.

In general, when an input to a gate or device contains an inverter, that input is said to be an active-low input, and requires a low (0) signal state to become activated or enabled.



Negative-AND			Negative-OR		
Inputs		Output	Inputs		Output
A	B	Y	A	B	Y
0	0	1	0	0	1
0	1	0	0	1	1
1	0	0	1	0	1
1	1	0	1	1	0

Figure 3-7 Negative-AND and Negative-OR Symbols and Truth Tables

Tri-state Buffer

While not truly a logic gate, the *tri-state*, or *3-state buffer* is an important component in the microprocessor system. It serves as a “switch” to either turn a signal on or off, see Figure 3-8. The signal at the active-low *enable* line \overline{E} enables the buffer when the signal is low, and disables it when the signal is high.

Figure 3-9 shows symbols for both active-high and active-low tri-state buffers. More than likely, however, you will see tri-state buffers incorporated into various components instead of by themselves. An example would be the data input/output lines within a memory chip. It's undesirable to have *all* RAM chips, for instance, connected or talking on the data bus at one time. This problem is solved by enabling only the data lines connected to the chip we wish to address by enabling that chip's data-line tri-state buffers and disabling the other chips' buffers. In essence, we are “disconnecting” the undesired chips temporarily.

Normally, a signal is in one of two states: high or low. The tri-state buffer actually forces the signal to a third high-impedance state when it is disabled, effectively removing the signal entirely. When the buffer is active, it behaves like a non-inverting straight-through circuit.

This subject will be discussed in greater detail in Section 5.

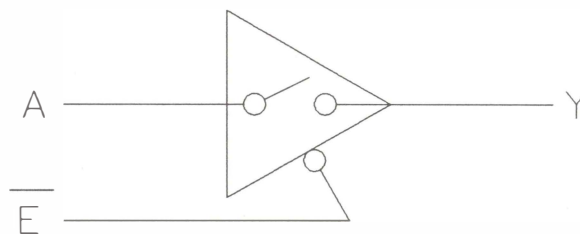


Figure 3-8 Tri-State Buffer Operation



Figure 3-9 Active-High and Active-Low Tri-state Buffers

Section 4

The Micro-Computer

Most of us have access to some sort of computer. We either have one in our home or use one at work. It usually consists of a box with a keyboard, mouse, and monitor attached. If all the pieces were itemized, the list might look something like this:

1. Main Console consisting of:
 - a. Microprocessor (CPU)
 - b. ROM (Read-Only Memory)
 - c. RAM (Random Access Memory)
 - d. Printer Port (I/O)
 - e. Communications Port (I/O)
 - f. Disk Drive (I/O)
 - g. Speaker (Output)
2. Keyboard (Input)
3. Mouse (Input)
4. Monitor (Output)

With so many and varied components, a working computer system is a remarkable thing to behold. By careful examination, however, everything in the list above falls neatly into one of the four categories described in the previous section. It is the microprocessor that oversees the control and synchronization of each part of the system and the system as a whole.

The Microprocessor

Since a microprocessor is a general-purpose device, there must be a way to instruct it to perform a particular application. Each type of microprocessor has a set of instructions which fall into the following categories:

- **Memory Read** (or Fetch) – Fetch data or the next instruction from memory
- **Memory Write** – Write data to memory
- **I/O Input** – Input data from an I/O device (e.g. keyboard, mouse)
- **I/O Output** – Display data on screen
- **Data Manipulation** – Add two numbers, compare two numbers, etc.

Through these *machine language instructions*, a microprocessor can be made to do numerous operations. Each instruction is represented by a binary code that can be stored in memory. A sequence of these coded instructions taken collectively represent a *program* to which the microprocessor responds by performing logical and arithmetic operations, by controlling the flow of signals to and from the system, and by routing these signals to their proper destination and in proper sequence to perform a particular task.

These sequences of instructions, or programs, can be stored for later retrieval and execution in many ways. Programs and data can reside in memory either within the microprocessor itself, or in space located in external memory such as RAM or ROM. They can also be stored on peripheral I/O devices such as disk and tape drives.

How is the microprocessor able to retrieve programs and data from such a variety of places? Furthermore, how does the microprocessor communicate with the I/O devices mentioned above? These questions bring us to the backbone of microprocessor-based systems: the microprocessor bus.

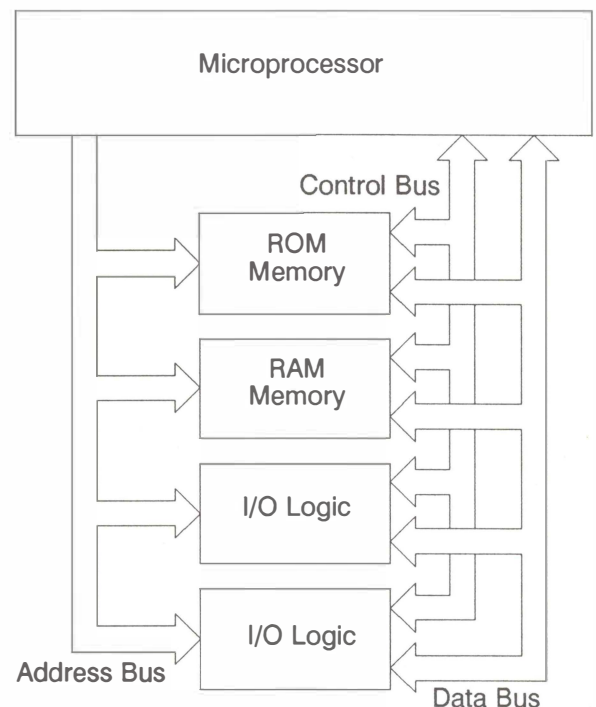


Figure 4-1 Typical Microprocessor Bus System

The Microprocessor Bus

In order for the microprocessor to control and pass information with and between the various I/O devices and memory, some form of communication must take place. This all takes place through the *microprocessor bus* (which is a set of conductive paths) that serves to interconnect two or more functional components of a system, Figure 4-1. These parallel signal lines are connected to each and every device (or chip) in the system that communicates with the microprocessor.

The bus is divided into three sections: (1) the *address bus*, which carries the location or *address* of the memory or I/O device the microprocessor is communicating with, (2) the *data bus*, which carries the actual information being communicated, and (3) the *control bus* used to keep everything timed and organized.

The address bus is *unidirectional*, that is, the data goes in only one way—from the microprocessor to the memory or I/O. The data bus on the other hand, is *bi-directional*. This allows data transfer to occur to and from the microprocessor. The control bus has both uni- and bi-directional lines.

The Address Bus

The microprocessor interacts with external devices such as memory and I/O devices by reading data from them and writing data to them. This implies that the microprocessor has some way of telling an external device that it wishes to communicate with it.

This is the purpose of the address bus, the logical name given to a specific group of signal lines which are output from the microprocessor. Each line is referred to individually as an address line.

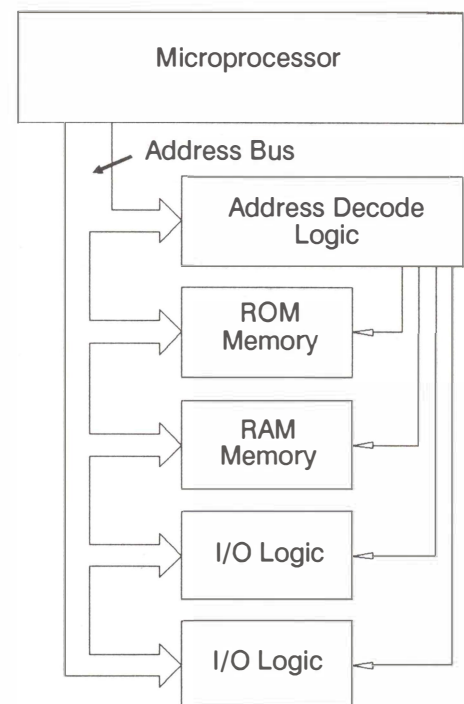


Figure 4-2 Address Bus Block Diagram

Address Bus Usage

The total number of discrete locations, or *addresses* to which a microprocessor can read or write is dependent on its *address bus width*, (determined by the number of address lines on its address bus). An address bus with a width of eight bits (eight lines) can *address* 256 locations, which can be any combination of memory and I/O addresses. A 16-bit wide address bus has 65,536 (64K) addresses. Most microprocessor systems address at least 64K of memory and subsequently employ an address bus at least 16 bits wide.

In memory, each address generally contains one *byte* (eight bits) of data. Let’s look at the arrangement of data on an example memory device, Figure 4-3. For argument’s sake, we’ll assume that our memory chip stores data one-byte across and has a capacity of 8K (or 2000₁₆ bytes) of storage. The first byte of data on our sample chip will reside at address 0 and the last byte at address 1FFF.

As we can see in Figure 4-3, address 0 contains the number 1A, address 1 contains F0, and so on. This chip requires a total of 13 address lines to address the entire 8K memory range, Figure 4-4. This is precisely how many address lines are in a typical 8Kx8 ROM chip. These lines are labelled A0 through A12 and are binarily weighted, with A0 weighted with a value of 1, A1 a value of 2, A2 a value of 4, etc.

Address	Data
0000	1A
0001	F0
0002	A5
0003	11
↓	↓
1FFD	00
1FFE	13
1FFF	EA

Figure 4-3 Example chip addressing

Address	Address Lines
0000	0 0000 0000 0000
0001	0 0000 0000 0001
0002	0 0000 0000 0010
0003	0 0000 0000 0011
↓	↓
1FFD	1 1111 1111 1101
1FFE	1 1111 1111 1110
1FFF	1 1111 1111 1111

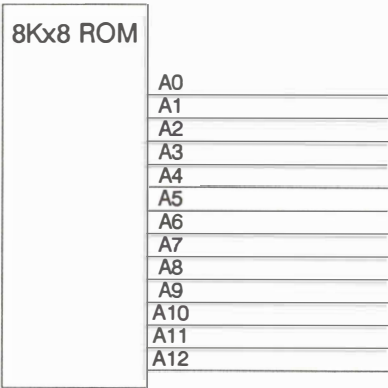


Figure 4-4 Number of Address Lines

Now suppose that there are four identical 8Kx8 memory chips in our system assigned the address ranges in Figure 4-5 below. Additional address lines and address decode logic are necessary because there are four devices on the bus, each capable of responding to the *read* signal asserted by the microprocessor. And since each chip can only respond to the lower 13 of the 16 address lines, the address decode logic and chip select lines provide the means to determine which device on the bus to address.

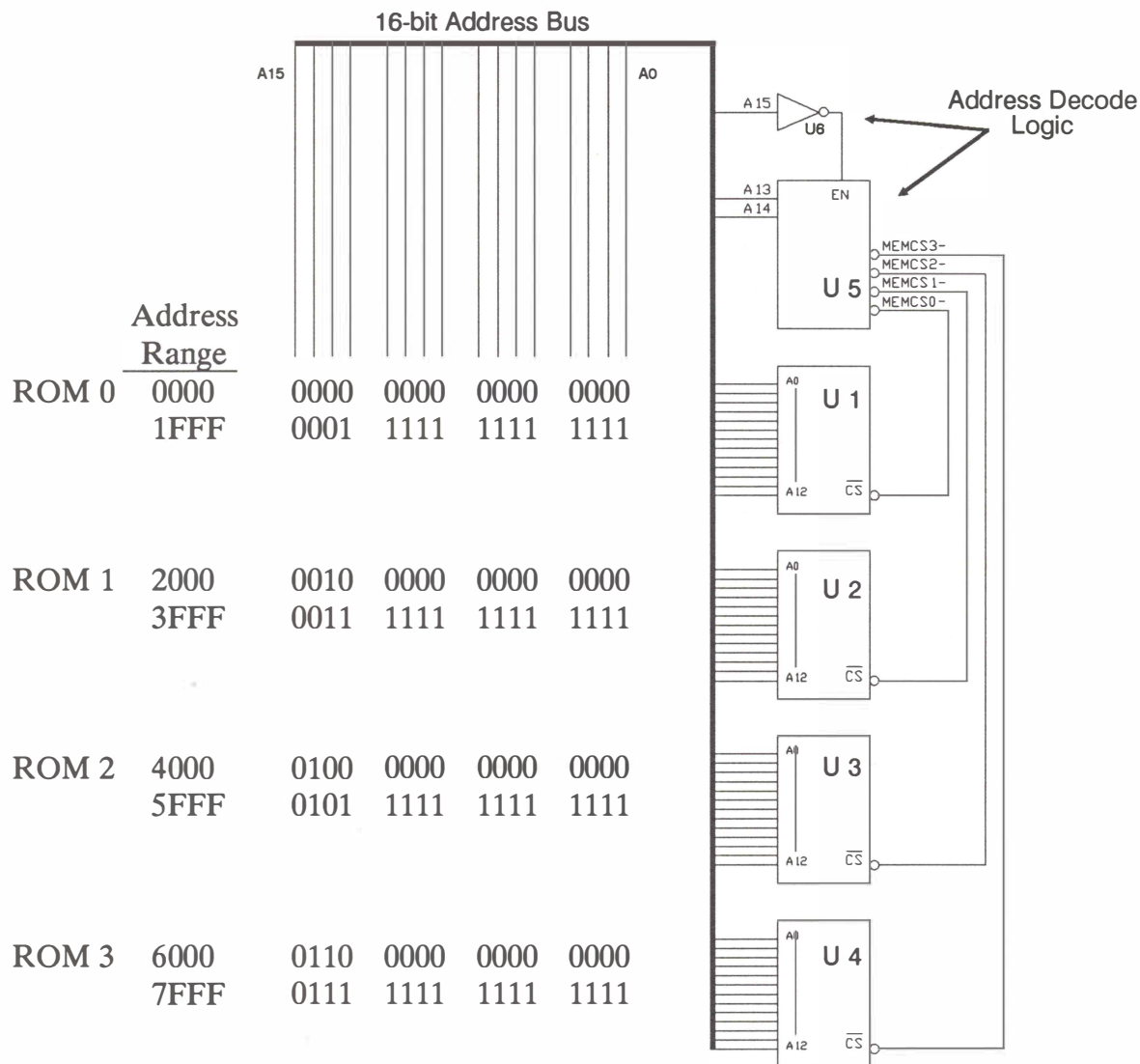


Figure 4-5 Addressing four ROMs

In Figure 4-5, U1 through U4 represent the four ROM chips and U5 with the inverter U6 comprise the address decode logic. Notice the three inputs to U5, the address decoder: A13, A14, and inverted A15. The inverter (U6) is necessary because we want the decoder *enabled* (high) only when we are addressing one of the four ROM chips, (when address line 15 is low, or at 0). When enabled, the function of U5 is to determine which of the four ROMs is being addressed based on the address lines A13 and A14. Each output of U5 goes to the *chip select* pin of its respective ROM chip and one and only one of these select lines is active (low) at any given time.

When memory in the range of 0 through 1FFF is addressed, the decoder places an active low on line MEMCS0 (the chip select line for ROM 0). Similarly, if memory in the range 2000 through 3FFF is addressed, MEMCS1 is placed active, and so on for the address range of each chip. Notice that it's not necessary to decode the lower 13 bits of the address bus in order to derive the proper chip select line. When a chip's chip select line goes active, the address on the bus at that time is within the assigned address range of that chip. The other ROM chips are tri-stated (disabled). In our example, address lines 0 through 12 are applied directly to the memory and are used to access the individual location within the selected memory chip.

If any address above 7FFF is accessed, address line 15 would be high, causing the decoder to be disabled by the inverter, in turn, causing each of the ROM's chip select lines to remain high, disabling the chip. Simply put, if A15 is low, the ROMs are being addressed. If it's high, the ROMs are turned off.

Number of Address Bus Lines

The number of address bus lines which a micro-processor can drive dictates the size (number of locations) of the address space. Figure 4-6 illustrates the relative address space of various bus widths.

Bus Width	Size of Address Space
8 bits	256 bytes
16 bits	65,536 (64 Kilobytes)
20 bits	1,048,576 (1 Megabyte)
24 bits	16,777,216 (16 Megabytes)
32 bits	4,502,585,344 (4 Gigabytes)

Figure 4-6 *Bus Capacities*

Memory Read

Now let's review what has been covered thus far. To transfer a byte of data from the memory to the microprocessor, a *read* operation must be performed:

- The address of the data to be read is placed on the address bus in a binary pattern.
- The address is then read by the *address decode logic* which examines the state of selected address lines (a "1" or "0") and determines (or *decodes*) which external device is being addressed.
- A single line (chip select) connected only to the device being addressed is made active to indicate that the microprocessor wishes to communicate with it.
- At the memory, the address bits are decoded and the desired memory location is selected.
- The microprocessor issues a *read* signal causing the contents of the selected address to be put on the data bus. The data byte is then loaded into the microprocessor's data register.

Now we'll examine the data bus in detail.

Data Bus

As stated earlier, the microprocessor is capable of only three basic action types:

- Reading from external devices (memory and I/O)
- Writing to external devices (memory and I/O)
- Data manipulation (addition, subtraction, comparison, etc.)

The microprocessor determines what external device it wishes to communicate with through the address bus. It must also have a way to transfer the data which is being communicated. This is the function of the data bus, a series of bidirectional signal lines connected from the microprocessor to each device that exchanges data with it, Figure 4-7.

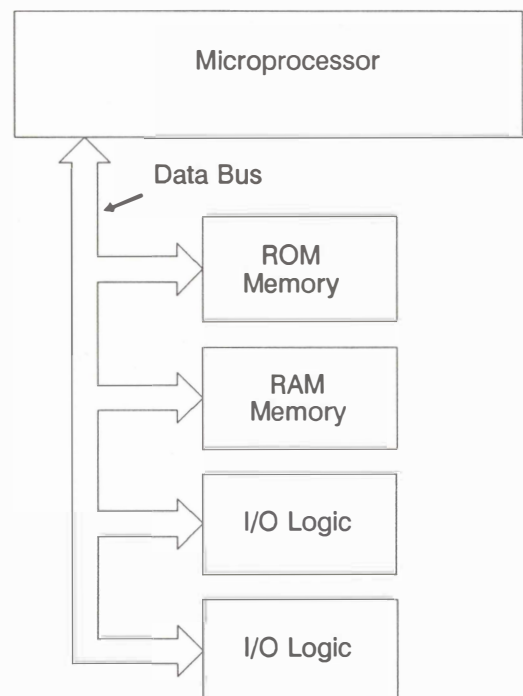


Figure 4-7 Data Bus Block Diagram

Since the data bus is bi-directional, and a variety of devices both “talk” and “listen” on the bus, some form of technique must be used to prevent data from more than one device being present on the bus at the same time. If a device is neither being written to nor read from, the data on its I/O lines is invalid, and if not disconnected or disabled, could interfere with valid data from another device.

As stated earlier, a signal line can be in one of three states: high, low, and tri-state, with tri-state equivalent to being “disconnected”. Since only one device other than the microprocessor should be talking or listening to the data bus at any given instance, all other devices connected to the data bus (other than the microprocessor) should be tri-stated. In fact, the tri-state, or *high-Z* state is the usual condition of the I/O lines of devices connected to the data bus. It is only when the device is explicitly addressed that the I/O lines are enabled.

Data Write Operation

In a typical write operation, the microprocessor orchestrates the following bus activity:

- Microprocessor places the device address on the address bus and strobes the *address latch enable* line to cause the address to be latched in the external *Address Latch*, a special buffer used to hold addresses.
- Microprocessor activates, or *asserts* the appropriate write control line to indicate to the addressed device that this is a *write* operation.
- After sufficient time has elapsed to allow the *address decode* circuitry to decode the address of the target device, the microprocessor places the data to be written to the target device on the data bus.
- After sufficient time has elapsed for the addressed external device to latch (read and store) the data from the data bus, the microprocessor ceases to drive the data onto the data bus and *de-asserts* the write control line.

Data Read Operation

When the microprocessor is attempting to read data from an external device, the following actions occur:

- Microprocessor places the device address on the address bus and strobes the *Address Latch Enable* to cause the address to be latched in the external *Address Latch*.
- Microprocessor asserts the appropriate read control line to indicate to the addressed device that this is a read operation.
- After sufficient time has elapsed to allow address decode to occur, the microprocessor reads the data output by the addressed device from the data bus into the appropriate internal register of the microprocessor.
- Microprocessor *de-asserts* the read control line.

At the moment which the microprocessor writes data to or reads data from the data bus, only the addressed device should be reading data from or outputting data to the data bus. If more than one device is attempting to exchange data with the microprocessor at any given instance, this indicates either a faulty design or logic failure.

Number of Data Bus Lines

Generally, the number of data lines present in a microprocessor's data bus is some multiple of eight, since the microprocessor deals in *bytes* of data. A microprocessor with a 16-line data bus is capable of dealing with data about twice as fast as one with only eight lines.

Some microprocessors have completely separate and distinct data and address busses. In other microprocessors, however (e.g. the 8085), a portion of the address bus serves a dual role. During the time when the microprocessor is outputting an address, these lines carry a portion of the address, while during the time that data is being either written or read these same lines serve as data lines.

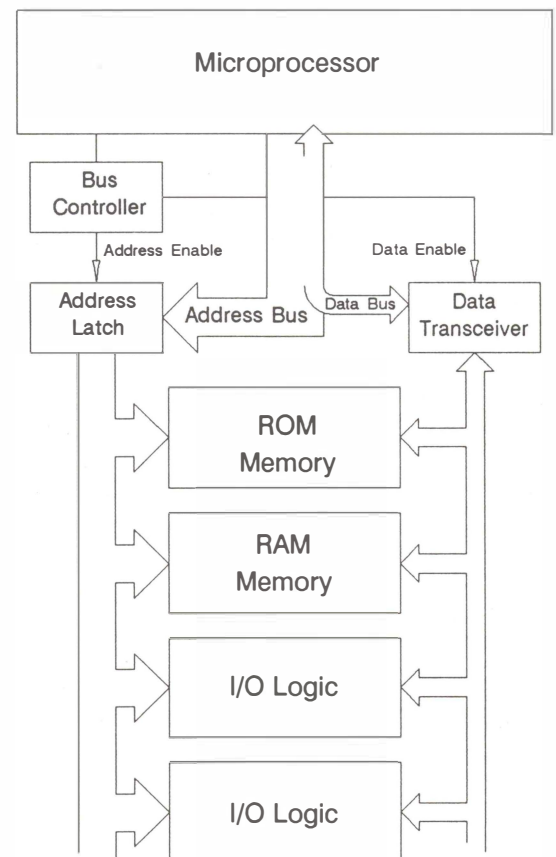


Figure 4-8 Multiplexed Address/Data Bus

This type of bus structure is said to be *time multiplexed* and was first implemented by Intel in its 8085 microprocessor. The advantage this scheme has is that it reduces the number of necessary pins on the microprocessor. The economy of multiplexing allowed the addition of interrupt capability to the 8080 (producing the 8085) without the addition of pins to the microprocessor package.

The width of the data bus and whether it's time multiplexed or not depends on the microprocessor family you're dealing with. Figure 4-9 lists some of the possibilities.

Microprocessor	Number of data lines
Intel 8080	8
Intel 8085	8 (time multiplexed)
Intel 8086	16 (time multiplexed)
Intel 8088	8 (time multiplexed)
Intel 80286	16
Intel 80386	32
Motorola 68000	16
Motorola 68020	32
Motorola 68030	32

Figure 4-9 Microprocessor Data Lines

Section 5

Memory

Microprocessor-based systems generally require vast amounts of storage to facilitate program execution and data usage in its operation. There are various types of storage used in such systems ranging from small, internal data registers capable of storing a single word (16 bits) within the microprocessor itself, to large-capacity storage devices capable of storing up to a million bits of data.

Semiconductor Memory

This type of storage is divided into two main types: *volatile* memory and *non-volatile* memory. Data stored in volatile memory is lost when power to that memory is removed or lost, while data stored in non-volatile memory is retained, even if the device is removed from its host system.

Read-Only Memory (ROM)

Read-Only Memory (ROM) is a form of non-volatile memory. It is memory from which data can be read, but not written. A ROM device is principally used to store programs not intended to be altered, such as a computer system's Basic I/O System (BIOS). Software that is stored on non-volatile devices such as ROM is called *firmware*, which is a somewhat logical halfway-point between *hardware* and *software*.

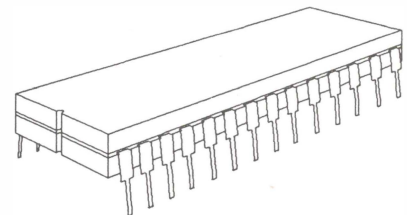


Figure 5-1 Read-Only Memory Chip

Programs or code embedded as firmware is difficult, if not impossible to change, depending on the type of device in which it exists. The data contained within the ROM, however, must initially get there somehow. Usually, the data is embedded into the device during manufacture by a *masking* process. These *mask* ROMs, as they are sometimes called, are not alterable once they are manufactured.

The Programmable ROM (PROM)

Another type of ROM device is called the *PROM*, or *Programmable Read-Only Memory*. It is manufactured without embedded data, and is *programmed* using a device called a ROM Programmer to embed data into the device. Once programmed, however, the data is permanent and cannot be altered. These devices are in reality, an array of *fusible links*. Data is written to them by selectively blowing, or *fusing* these links open. A fused link represents a “0” and an unfused link a “1”. After being fused, a link cannot be restored, therefore, PROMs can only be programmed once.

Once programmed, PROMs offer a great deal of security from inadvertent erasure or alteration. In general use, a PROM cannot be altered. Only a ROM Programmer has the necessary circuits and current to program a ROM.

The Erasable Programmable ROM (EPROM)

The *Erasable Programmable Read-Only Memory* or EPROM is yet another non-volatile storage device. These chips may be erased and subsequently reprogrammed and are, therefore, reusable. They are easily recognizable by a clear quartz crystal window in the middle of the top surface of the chip. This clear window exposes the semiconductor memory inside. By exposing the memory to a special high-intensity ultraviolet light for a certain length of time, any data previously stored is erased.

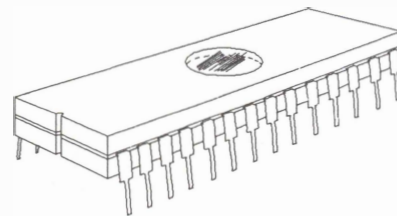


Figure 5-2 Erasable Programmable ROM Chip

Each storage location (bit) in an EPROM is comprised of a capacitor. The state of the capacitor, (the presence or absence of a charge), determines the existence of a “1” or “0” in that bit position. To erase the chip, the chip’s quartz window is exposed to a special UV light, thereby exciting the electrons within the capacitors comprising the memory. After enough time has elapsed and the electrons have become sufficiently excited, the capacitors discharge, thereby clearing memory. The chip can then be programmed using a device called a *PROM programmer*.

The Electronically Erasable PROM (EEPROM)

The last ROM device we’ll discuss here is the *Electronically Erasable Programmable Read-Only Memory*, or *EEPROM*. Another name for this device is the *EAPROM*, for Electronically *Alterable* PROM. These devices can be erased and reprogrammed while still installed in the microprocessor system. These devices are fairly expensive in comparison to ROMs, PROMs or EPROMs and are not very common for that reason.

ROM Data Organization

ROM data are usually organized in bytes, that is, for any given ROM address, there exists a byte of data. A typical 64K (8Kx8) ROM, the 2764, is illustrated in Figure 5-3. When a 13-bit address is present at address lines A0 through A12 and the chip enable lines \overline{G} and \overline{E} are low, an eight-bit data output is present at Q1 through Q8. This is an example of a medium-capacity ROM device. ROMs capable of storing as few as 1024 bits exist in applications with small memory requirements.

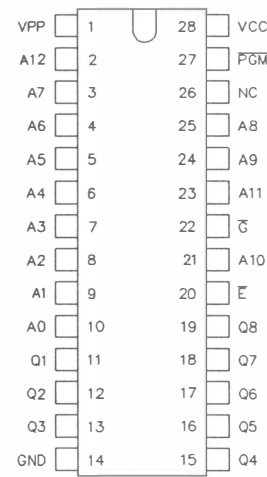
The data lines Q1–Q8 in Figure 5-3 are *tri-state* outputs enabled by \overline{G} . As discussed in Section 3, memory devices usually have tri-state data output lines to prevent signals from more than one device existing on the data bus at the same time.

Access Time

The time that it takes for the ROM to produce the contents of a given address to the data bus once that address has been provided to the chip is called the chip’s *access time* and is measured in microseconds, (μ s). Access time alternatively is measured from the time the chip select line is enabled and the occurrence of valid output data if a valid address is already present.

Figure 5-4 illustrates access time for a typical ROM device through a *timing diagram*. The access time t_a is measured from the instant a valid address is present at the address bus to the instant valid data is present at the data bus. The chip’s chip select line is assumed to have been enabled.

Typically, faster CPUs require correspondingly faster memory devices, otherwise, performance degradation results. Memory devices, therefore, are usually available in a choice of access times to match processor speed.



Pin Nomenclature	
A0-A12	Addresses
\overline{E}	Chip Enable
\overline{G}	Output Enable
GND	Ground
NC	No Connection
PGM	Program
Q1-Q8	Outputs
VCC	+5-V Power Supply
VPP	+21-V Power Supply (during programming only)

Figure 5-3 2764 EPROM Chip

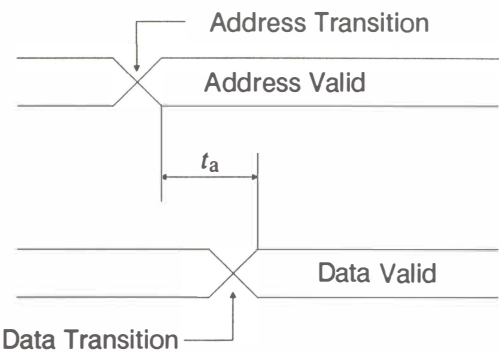


Figure 5-4 ROM Access Time

ROM Usage

When using ROMs in a system with an eight-bit microprocessor, data located at ROM addresses correspond to system addresses on a one-to-one basis. In a 16-bit environment a slightly more complex arrangement is necessary. As in the eight-bit system, each address contains one byte of data, however, the microprocessor deals with data 16 bits at a time. In fact, the 16-bit microprocessor for the most part, deals with only *even* addresses. When any given even address is accessed, the next sequential odd address is accessed as well.

Figure 5-5 shows how a high bank and low bank of ROM is interfaced to a 16-bit data bus structure. The address bus lines A1 through A13 are connected to the ROM address pins A0 through A12. The High Bank ROM's output lines O0 through O7 are connected to the data bus lines D8 through D15 and the Low Bank ROM's output lines O0 through O7 are connected to the data bus lines D0 through D7.

Whenever any *even* ROM address is accessed, the Low Bank ROM supplies the least-significant byte of the 16-bit data word and the High Bank ROM supplies the most-significant byte. Each byte resides in the same physical address of their respective chip. What this scheme does is access two ROM devices simultaneously, one at an *even system* address, and one at an *odd system* address.

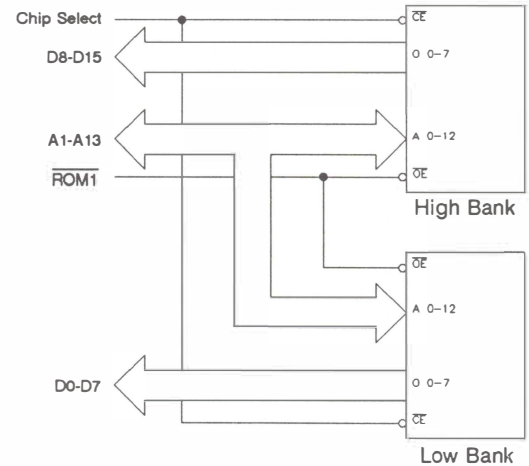


Figure 5-5 ROM/PROM/EPROM Bus Interface

Random Access Memory (RAM)

Random Access Memory (RAM) is a form of semiconductor storage from which data can be readily written to as well as read from at any selected address in any sequence. Any data written to a RAM location replaces the data previously stored there, and remains at that location as long as power to the chip is maintained or is overwritten by subsequent data. Reading data from a RAM address takes place through a *nondestructive read* operation and does not alter its contents—the data is merely copied from the RAM address to its destination.

Since interruption of power to the RAM device will result in loss of any stored data, RAM is a form of *volatile* memory.

Static vs Dynamic RAM

Static RAM is comprised of a network of latches and dynamic RAM is composed of simpler and faster capacitors. Static memory retains its data as long as power is applied to the chip. Dynamic memory, in utilizing simpler circuitry, is less expensive to produce. However, due to the nature and size of the capacitors used, it cannot retain its data without periodic *refreshing*. This refreshing can be thought of as a “topping off” or “recharge” and is accomplished through circuitry external to the memory chip itself. Whenever DRAMs are used in a microprocessor system, this refresh circuitry must be considered, adding to the complexity of the overall system design.

Each memory cell requires refreshing every two to four milliseconds accounting for a significant percentage of bus traffic. The microprocessor is prevented access to the memory’s address lines during these refresh cycles, and is held in a state of suspension, called a *wait-state*. In systems incorporating large amounts of memory (one Mbyte or more), these DRAM refresh cycles significantly reduce microprocessor throughput by asserting wait-states.

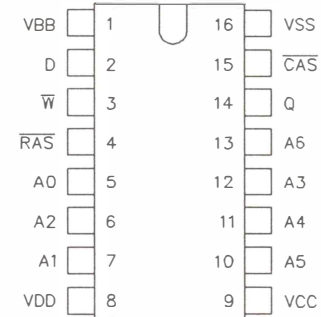
Since static memory requires no such refresh operation, it is sometimes used as special *cache memory*. Even though it is a slower memory compared to dynamic RAM, when used for memory-intensive applications, its independence to refresh yields substantial performance increases. This type of memory is sometimes referred to as “zero-wait-state” memory.

4116 Dynamic RAM Chip

The 4116 Dynamic RAM (DRAM) chip is shown in Figure 5-6. It has a 16,384 X 1 bit architecture addressed by the seven address lines A0 through A6. At first, you may think this strange. How can seven address lines access 16,384 addresses? The total number of addresses obtainable with seven lines is only 128! In addition, what are the *row* and *column address strobe* lines? These lines were not present on the ROM devices studied earlier.

To understand the 4116’s addressing scheme, and the addressing of other RAM devices, it’s necessary to look at how the storage cells within the chip itself are configured.

The cells within the 4116 are not actually arranged in one long string of bits as the 16,384 X 1 specification may indicate. The cells are arranged in a 128 X 128 matrix, or grid, Figure 5-7. To address a particular cell within this grid, a *column address* and *row address* are needed. First, the row address is applied to the address lines of the chip and the *Row Address Strobe* ($\overline{\text{RAS}}$) line is strobed, latching the row address into a *row address buffer*. The column address is then applied to the same address lines and the column address is strobed loading the column address into its buffer. The chip’s internal address decoder can then access the particular memory cell, extract its contents and output it at the data out pin, Q.



Pin Nomenclature	
A0-A6	Addresses
CAS	Column Address Strobe
D	Data Input
Q	Data Output
RAS	Row Address Strobe
VBB	–5-V Power Supply
VCC	+5-V Power Supply
VDD	+12-V Power Supply
VSS	Ground
W	Write Enable

Figure 5-6 4116 Dynamic RAM Chip

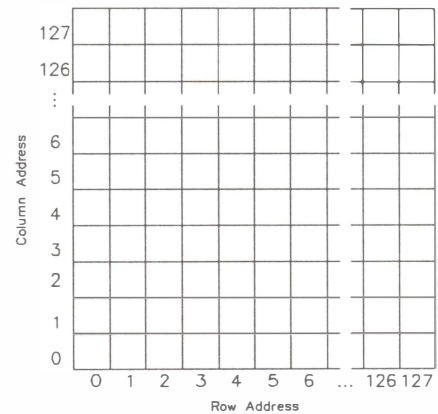


Figure 5-7 Memory Cell Matrix of a 4116 DRAM Chip

Address Multiplexing

Since a RAM cell's *physical* address is divided into two parts, some form of juggling must take place to convert the system address to the required row and column address. The number of system address lines required to address 16,384 cells is 14. The number of address lines on the chip is seven. The lower-order seven address lines can be used as the row address and the higher-order seven lines used as the column address. An address *multiplexer* is used to split the 14-bit address into the two seven-bit row and column address required by the chip, Figure 5-8.

The multiplexer splits the system address at its input into two discrete addresses. It first places the low, or row address (from address lines A0–A6) on the multiplexed address bus lines RA0–RA6 and strobes the RAS line. The high, or column address (from address lines A8–A13) is then placed on the multiplexed address bus and the CAS line is strobed.

Data Organization

Unlike the ROM devices discussed earlier, RAM devices store one bit of data per chip location in one long string of contiguous bits. Typically, the microprocessor expects each memory location to contain a byte (eight bits) of data. This means that RAM must be arranged in such a fashion that eight RAM devices can be addressed in parallel (simultaneously) to allow the reading or writing of a byte at each requested address.

In our example in Figure 5-8, the single data output of each RAM chip is connected to its respective data line on the data bus. Each RAM's address lines are connected in parallel to the multiplexed address lines coming from the address multiplexer. Through this multiplexing, this bank of RAM provides a byte of data at each address.

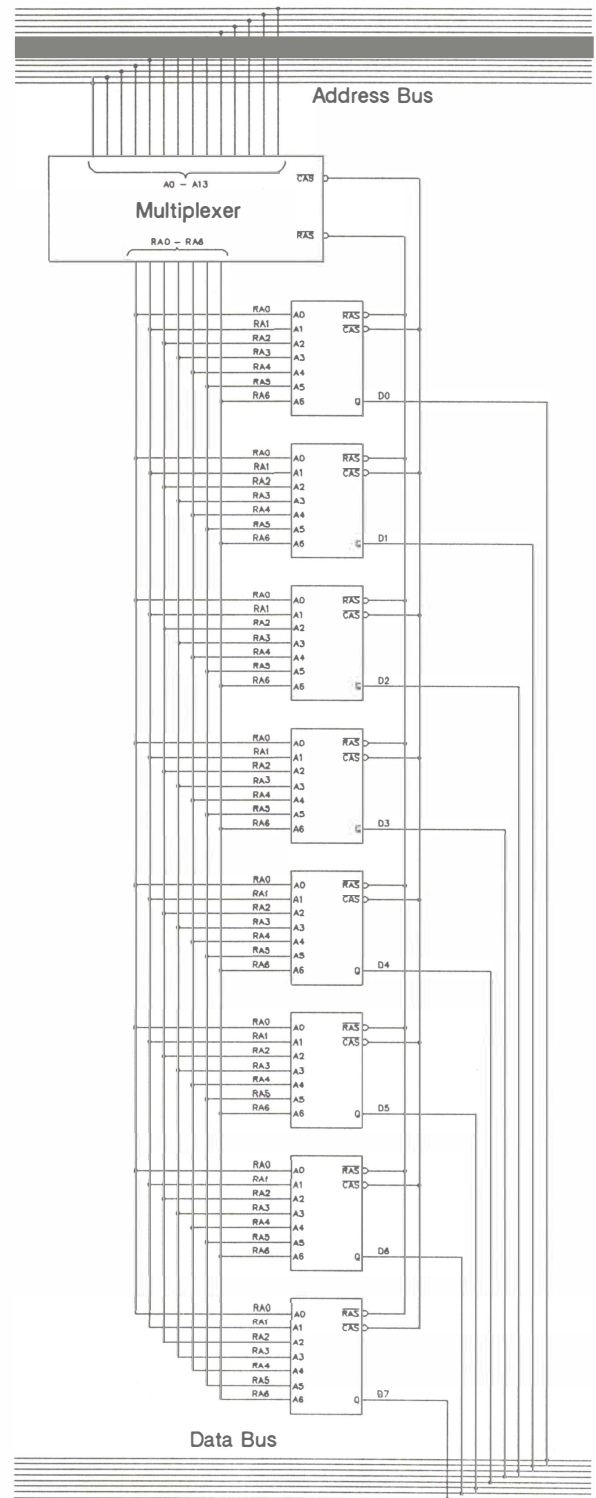


Figure 5-8 RAM Address Multiplexing

RAM Usage

16-bit microprocessors access memory in increments of 16 bits. 32-bit microprocessors, predictably, access 32 bits at a time. Memory, however, is arranged with one byte per address, so a 16-bit microprocessor must deal with two addresses at a time when reading or writing to memory. Likewise, a 32-bit microprocessor must deal with four.

Figure 5-9 shows how data is addressed in a typical 16-bit system. Notice that the data is accessed through *even* addresses only. Also notice that the least-significant byte of each word resides at the even address and the most-significant byte at the odd address.

A scheme similar to the one used in Figure 5-5 for ROM addressing is used in RAM addressing in a 16-bit system. Memory is physically arranged in groups of eight chips called *banks*. Some systems employ a ninth *parity* chip for error checking in each bank.

Each bank handles one byte of data at a time. Two banks, therefore, are necessary to accommodate a 16-bit word. One bank is assigned to the low byte (even address) and the other to the high byte (odd address). Each 16-chip bank (high and low) corresponds to a particular address range with the size of the range dependent on the capacity of each of the DRAM chips.

The memory devices used in Figure 5-10 are 256Kx1 DRAM chips arranged in four *banks* with each bank consisting of two groups of eight chips. The lower eight chips are connected to data lines D0 through D7 (least significant byte, even address) and the upper eight chips to D8 through D15 (most significant byte, odd address).

Address	Data
⋮	
FEFF	
FF00	D 5
FF01	A 6
FF02	0 1
FF03	C B
FF04	2 2
FF05	6 3
FF06 = 1B4E	4 E
FF07	1 B
FF08	7 9
FF09	E E
FF0A	D 5
FF0B	A A
FF0C	F F
FF0D	F F
FF0E	0 0
FF0F	0 A
FF10	
⋮	

Figure 5-9 Word Addressed Data

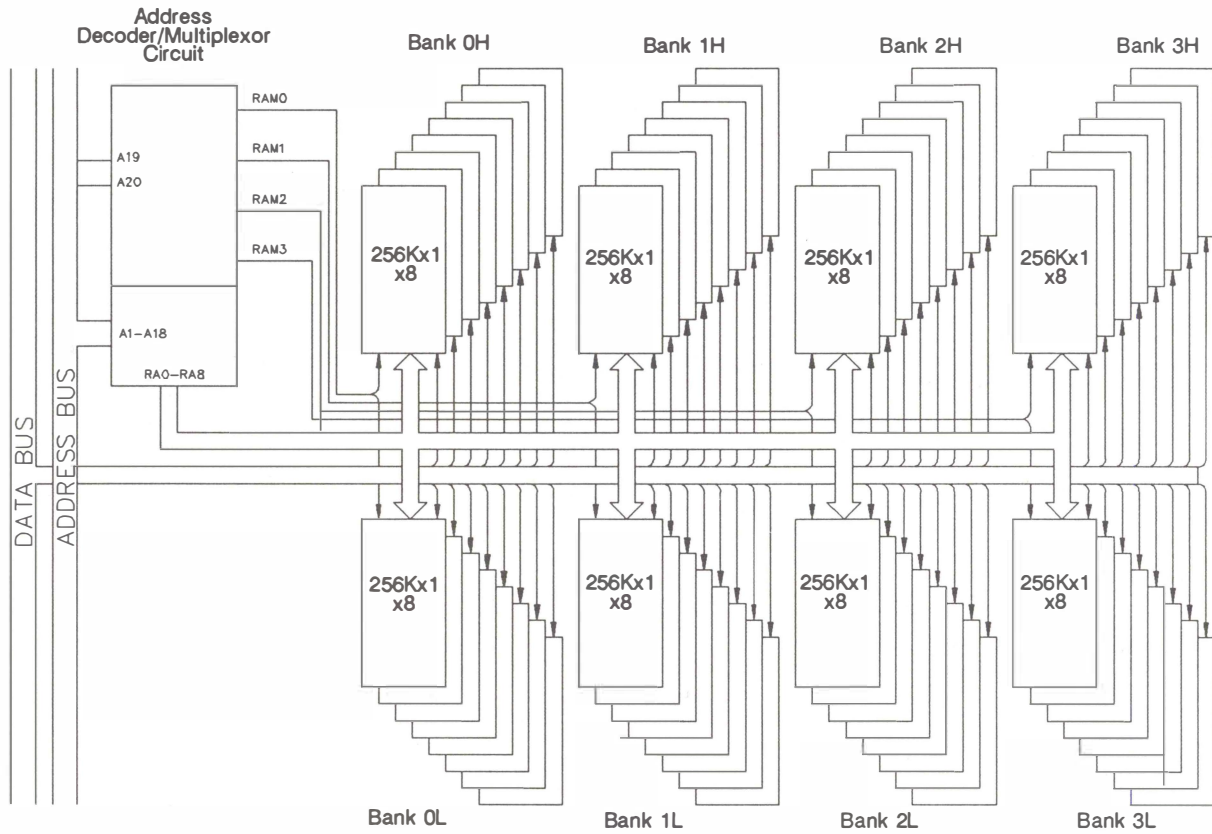


Figure 5-10 2 Megabyte RAM Expansion

The Decoder circuitry determines which bank of memory to access by interrogating lines A19 and A20. Address lines A1 through A18 are multiplexed into a nine-bit row and column RAM address and output to the RAM-Address bus lines RA0 through RA8. By using 256Kx1 chips in four 16-chip banks, 2 megabytes of addressable memory is realized.

Refresh Circuitry

As stated earlier, dynamic RAM must be refreshed every 2 msec or so to properly maintain its data. This refreshing must be done by devices external to the memory chips themselves so for any microprocessor-based system utilizing DRAMs, refresh circuitry must be designed into the system.

Refresh is accomplished by strobing the $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ line for each row or column address in each RAM chip. A 16Kx1 DRAM would require 127 iterations to completely refresh all of the cells in its 127x127 cell matrix, whereas a 256Kx1 DRAM would require 512 iterations. In most systems, refresh is not accomplished

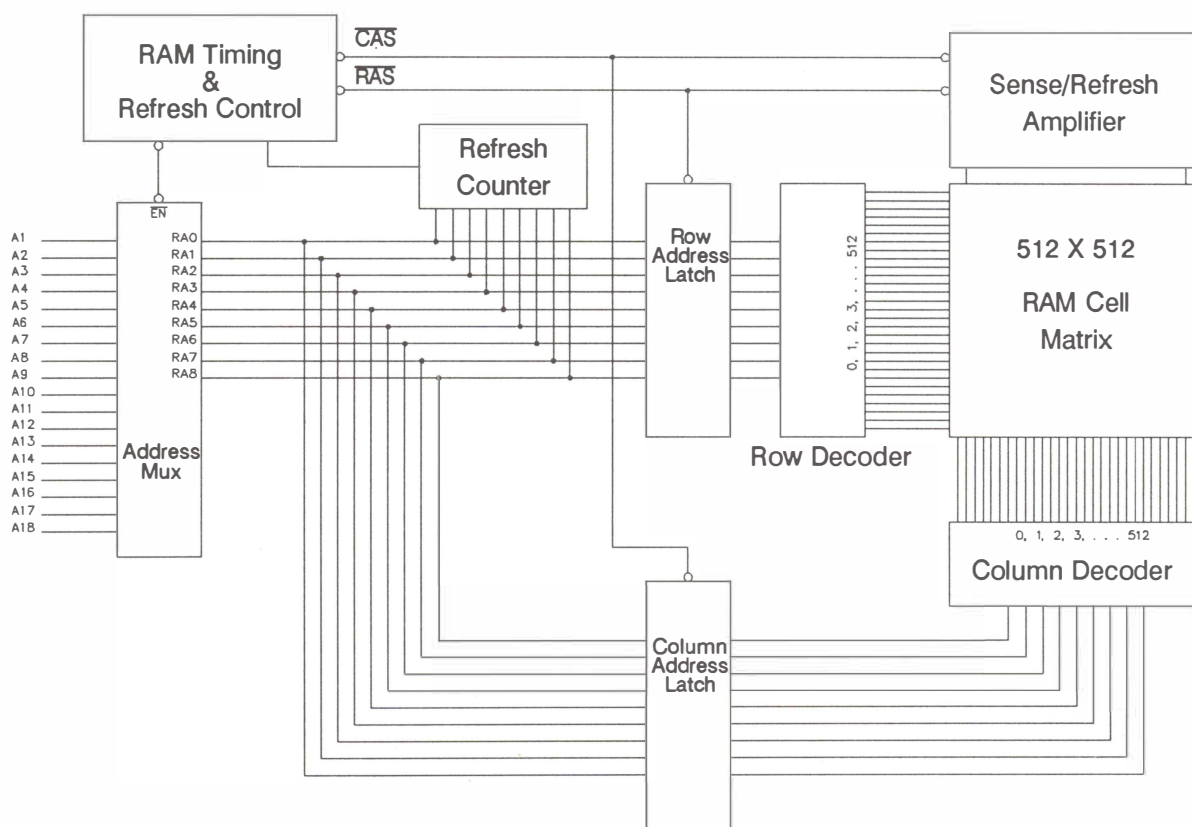


Figure 5-11 Typical RAM Refresh Circuit

all at once. It is interlaced between CPU cycles and occurs only when no RAM access is taking place. Likewise, when refresh is taking place, the CPU is forced to wait until the current refresh cycle is finished. A *refresh counter* is maintained to keep track of which rows have been refreshed and when all 512 rows of a 256K DRAM chip, for instance, have been refreshed, resets to zero and starts counting again.

Figure 5-11 shows a simplified refresh circuit using the RAS-only refresh method. The RAS and RAM address lines are connected in parallel to all the RAM chips in the circuit so in any given refresh cycle, all RAM is refreshed simultaneously. In other words, if the refresh counter was at row 156, then row 156 of every RAM chip on the bus would be refreshed during that refresh cycle. Regular row and column address line multiplexing is disabled during refresh so that the refresh row address can be applied to the RAM address lines from the refresh counter.

In an actual circuit, refresh may be accomplished in a number of different ways. The basic object is to strobe each row or column once every two to four milliseconds either all at once (*burst* refresh) or in between processor memory accesses (*interlaced* refresh).

Section 6

Input/Output

Communication with the outside world is handled through the microprocessor's Input/Output, or I/O capability. As with memory devices, communication with I/O devices such as video displays, printers, and keyboards is handled via system addresses.

What is an I/O device?

An I/O, or *peripheral* device is any external device that exchanges data with the microprocessor. This communication can be bi-directional, as in a disk drive or modem, or uni-directional, as in a keyboard or video display. In much the same way as we write to and read from memory at discrete addresses, peripherals communicate through I/O *interface* devices at discrete addresses as well. These interfaces are often referred to as I/O *ports*, or *channels*.

Where memory devices store and present data, interface devices send and receive data, depending on the configuration of the port. Certain interfaces are flexible in that they may be programmed to behave as input ports, output ports, or both an input and output port. These *Peripheral Interface Adapters* (PIAs) in turn can drive external display devices such as LEDs, or read stimuli from external devices such as joysticks or switches.

I/O Addressing

I/O ports are located at system addresses in much the same way as ROM and RAM. Writing to an I/O port involves a software command that puts an address on the address bus and sends data across the data bus, as in writing to memory.

There are two different addressing schemes used for I/O address mapping. In one, a single contiguous address range is divided into RAM, ROM, and I/O address space, called *Memory-Mapped I/O* addressing. The other, called *I/O-Mapped* addressing, maintains two separate and distinct address ranges: one for memory and one for I/O.

Memory-Mapped I/O

Memory-mapped I/O is used by the Motorola family of microprocessors and is the more straightforward of the two addressing schemes. The MC68000 microprocessor, for example, has the capacity to access 16 megabytes of address space in which no distinction between memory and I/O communications is made. The I/O ports themselves and their associated circuitry provide the required distinction.

Figure 6-1 shows how memory-mapped I/O might be configured in a typical 68000 system. It is the responsibility of external circuitry to decode the address to determine whether the current operation is a memory or I/O access.

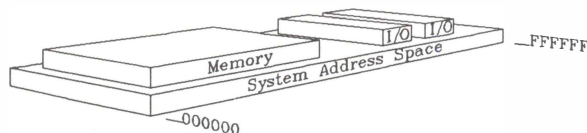


Figure 6-1 *Memory-Mapped I/O*

I/O-Mapped Addressing

The Intel family of microprocessors uses separate I/O-Mapped address space, Figure 6-2. This means that the microprocessor has a full range of addresses assigned to memory and a separate and distinct range of I/O addresses. There may be, for instance, a memory address 1A and an I/O address 1A. The two locations are completely separate from one another. The differentiation between the two is made through software by either using IN and OUT instructions for I/O reads and writes or the MOV instruction for memory reads and writes.

The microprocessor interprets the software instruction and asserts the required control lines: it first places the address on the address bus, then, depending on the type of instruction being executed, will assert the $\overline{M/\overline{IO}}$ line low for an I/O operation. Conversely, it is asserted high for a memory operation. Additional *status* lines, S0 and S1 for example, identify the beginning of a bus cycle and help identify the type of cycle—read or write. Through the $\overline{M/\overline{IO}}$, S0, and S1 control lines the bus controller chip is able to select the appropriate I/O or memory operation. In the mean time the $\overline{M/\overline{IO}}$ and address lines provide input to the address decode logic enabling it to select the appropriate I/O or memory device.

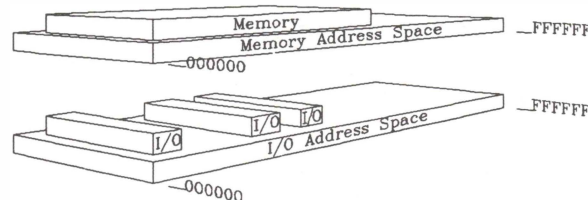


Figure 6-2 I/O-Mapped I/O

I/O Interfaces

A variety of special-purpose chips exist to facilitate interfacing peripherals to the microprocessor. These chips handle the multiplexing and buffering of data between the data bus and individual ports.

The Peripheral Interface Adapter (PIA)

The *Peripheral Interface Adapter*, or *PIA*, is a general-purpose interface device that allows the microprocessor to communicate with I/O devices and peripherals and comes in a variety of configurations. It manages the incoming and outgoing data, and like a memory device, occupies a particular address space.

Most PIAs can be programmed, or configured as either input ports or output ports, or input/output ports. A typical PIA as illustrated in Figure 6-3 might have three ports A, B, and C at addresses F8, F9, and FA respectively. A special fourth address FB is used to program, or configure the PIA.

During a power-up (boot) or reset, the ports are configured by writing data to address FB. Remember that in an I/O-Mapped system the boot program would **OUTPUT** to the address, whereas in a Memory-Mapped system it would simply write to the address. The actual data written is unimportant at this time, however, it's worth mentioning here that its content determines which ports are configured as input, output, or both.

It is then the responsibility of the application software program to read from and write to the ports. A typical routine might write to Port C to enable the reading of a set of switches, then continuously read from Port B (a keyboard) and display the result by writing to Port A (an LED display).

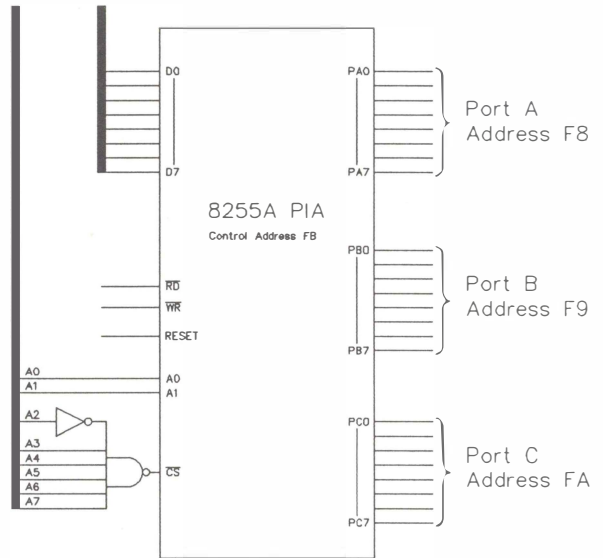


Figure 6-3 8255A PIA

Asynchronous Communications

Another device used to help the microprocessor communicate with the outside world is the *asynchronous communication* port. An RS-232 serial port is a popular example. Like the PIA, it resides at a particular system address and manages the input and output of data at its port. This device comes in many forms, the *Universal Asynchronous Receiver/Transmitter*, or *UART* being the most common.

The UART accepts parallel data from the microprocessor and converts it into a continuous serial data stream for transmission. It also receives serial data and converts it into parallel data to be read by the microprocessor.

In addition, the UART is able to alter data transmission rates, communications protocols, and perform required handshaking with external peripherals.

Other Interface Devices

Other interface devices include floppy disk, keyboard and video controllers, to name a few. In systems incorporating many such devices, address decoding circuitry may be necessary to provide chip select signals to the array of I/O devices on the address bus.

Bus-width Considerations

Most peripheral devices are byte-oriented. In other words, they are capable of only handling data eight bits at a time. When interfacing 8-bit devices to a system with a data bus eight bits wide, no special consideration is required. The eight data lines are simply connected to the eight input/output lines of the interface. When connecting such devices to a 16- or 32-bit system, special attention must be paid to which data lines are connected to the interface. In addition, the software written to control these ports must take into account their eight-bit configuration.

Usually in such systems, only the lower order byte (D0-D7) is connected to I/O channels and byte-mode reads and writes or inputs and outputs are performed on the lower half of data words.

Section 7

Interrupts & Direct Memory Access

Interrupts

In a typical microprocessor system, there may be dozens of devices attached to the microprocessor, any or all of which may need service by the microprocessor at any time. One way of handling this would be to constantly read the status of all the devices to determine if any require the services of the microprocessor.

This method, called *polling*, was widely used in earlier computers but has fallen out of favor due to its extreme inefficiency. Most of the CPU's time is eaten up polling the devices for a service request with very little time left over to get the real work done.

Due to this inefficiency, having the devices themselves “tap the shoulder” or *interrupt* the CPU was conceived. In this way, the CPU could concentrate on the main task at hand and service the external devices only when an *interrupt* occurred.

Most microprocessors that implement interrupts have an *Interrupt Request* pin on the microprocessor. When an external device like a keyboard needs to be serviced by the microprocessor to process a keystroke, for instance, it asserts its Interrupt Request line. This line is connected to an *interrupt controller* that has the responsibility of keeping track of what devices need serv-

icing as well as their relative priority should more than one device require servicing at one time. In addition, the interrupt controller maintains a table of *interrupt vectors* that point to the location within a table in memory where the microprocessor can find the starting address of the interrupt servicing routine written expressly for each device. These routines, or *interrupt handlers* are located in yet another part of memory.

When the microprocessor has completed the instruction it was currently executing, it acknowledges the interrupt and reads from the interrupt controller an interrupt vector. The microprocessor then stores the address of the current instruction in a safe place, like a “bookmark”, in memory or in an internal register.

It then takes the interrupt vector read from the interrupt controller and uses it to fetch the starting address of the service routine written specifically to service the interrupting device. The microprocessor then loads its instruction pointer register with that address and executes the service routine, which in our example, would be the interrupt service routine which reads a character from the keyboard and saves it.

Having completed its task, the instruction pointer would be re-loaded with the previously saved “bookmark”, and program execution would resume from where it was interrupted.

Types of Interrupts

There are different types or classifications of interrupts, varying from microprocessor to microprocessor. However, the following general classifications are fairly universal.

As we have seen, some interrupts are caused by external, hardware-related events, while others may be caused by the execution of a certain classification of instructions, or by the occurrence of some exceptional condition.

Hardware Interrupts

Non-Maskable Interrupts

Some types of microprocessors have an NMI (Non-Maskable Interrupt) pin. If an active signal is asserted on this pin, the execution of instructions by the microprocessor will be interrupted and the appropriate interrupt service routine will be executed.

This is called a Non-Maskable Interrupt because the microprocessor may not “mask out”, or disable, the recognition of the active signal on the NMI pin. NMI is typically used to signal critical or “catastrophic” types of external events to the microprocessor, such as imminent loss of power, memory error, or bus parity error.

Maskable Interrupts

Maskable Interrupts are applied to the microprocessor via the Interrupt Request pin. This is called a “maskable” interrupt because the programmer may choose to disable or “mask out” recognition of an active signal on the Interrupt Request pin.

An example of using this masking feature would be to disable the maskable interrupts until a critical portion of a program has completed execution.

Software Interrupts

Software Interrupts allow the programmer to call special Interrupt Service Routines without knowing where they reside in memory. As an example, the Intel INT instruction allows the programmer to execute an Interrupt Service Routine by its location in the Interrupt Table. An INT 21 instruction, for instance, would cause the microprocessor to perform the following tasks:

- Save the current contents of the Instruction Pointer in memory
- Load the Instruction Pointer with the 21st Interrupt Service Routine Pointer which usually resides in the lowest part of memory, beginning at address 0
- Transfer control to the Interrupt Service Routine pointed to by the Instruction Pointer
- When the Interrupt Service Routine completes execution, the Instruction Pointer is re-loaded with the previously saved contents
- Program execution resumes where it left off

The most commonly used implementation of software interrupts are the BIOS (Basic I/O System) calls in PC-DOS. Utilizing software interrupts to invoke the BIOS subroutines means the programmer doesn't have to be concerned with the exact address of the subroutine in memory. This means that a particular interrupt vector will point to the same Interrupt Service Routine regardless of the version of firmware, allowing for transparent and painless firmware maintenance.

Exceptions

A separate category of interrupt, the *Exception*, is worth mentioning here. Two different definitions exist, depending on whether you consult Intel or Motorola. Intel treats exceptions separately from interrupts, while Motorola categorizes all interrupts and errors as exceptions.

In either case, the principle difference, as far as troubleshooting microprocessor-based systems is concerned, is that exceptions are caused by unusual, unexpected conditions *found by the microprocessor during the course of program execution*, and interrupts are caused either by the execution of a software interrupt command or by an external device requesting service.

Exceptions are not always the direct result of a software glitch. It is possible for an exception to occur because of a hardware failure. The microprocessor just *thinks* its an exception. If a component on the data bus, for instance, is faulty, it would be possible for a divide-by-zero operation to be executed, causing an exception. Likewise, an erroneous opcode could be generated by a faulty bus resulting in an exception.

Direct Memory Access

Direct Memory Access is the term used to describe a situation during which a device other than the CPU seizes control of the address bus, data bus, and control lines and uses them to transfer data between a device and memory. DMA, however, only occurs between CPU memory or I/O accesses when the busses are not being used by the CPU.

When the CPU handles the transferring of a block of data from a device to memory (or vice versa), a software program is required to effect the transfer. It performs the transfer by executing a series of instructions repeatedly until the transfer is complete. This process is extremely CPU-intensive and monopolizes the CPU for a rather simple operation; like using a dump truck to move a shovelful of dirt.

The DMA Controller

Offloading this task to a *Direct Memory Access Controller* frees the CPU to do other more rewarding tasks. The DMA Controller typically comes in the form of a separate LSI component such as the Intel 8237 DMA Controller chip. This type of chip usually contains a number of independent DMA channels, each capable of handling a DMA transfer. The controller handles the necessary decrementing (or incrementing) of byte counters, the proper assertion of read and write lines, and general traffic control.

Since DMA occurs while the CPU is off doing things other than memory or I/O accesses, this interleaving of bus activity greatly increases the overall system efficiency.

Types of DMA Transfers

DMA falls into three categories:

- Memory-to-I/O transfers
- I/O-to-Memory transfers
- Memory-to-Memory transfers

The first two types involve no intermediate storage of the data between the origin and destination devices. One device *writes* to the data bus while the other device simultaneously *reads*. This process is extremely fast and offers a great improvement over CPU-orchestrated data transfer. Memory-to-Memory transfers, on the other hand, require that the DMA Controller temporarily store the data between the reads and writes since two separate addresses cannot exist on the address bus simultaneously.

Typical Uses for DMA

DMA is used in microprocessor systems for a multitude of tasks:

- It is used to refresh RAM memory by automatically strobing each of the rows in each RAM chip at least once every 2 ms.
- It is used in the transfer of data between memory and the disk controllers which in turn store and retrieve the data in disk sectors on either hard or floppy disks.
- It is used to transfer blocks of data from one part of memory to another.

To execute a DMA transfer, the CPU instructs the DMA Controller on where to find the data to transfer, the destination of the transferred data, and how many bytes to transfer. It then continues on with its program and lets the DMA Controller “steal” bus cycles between its own accesses to effect the transfer.

Section 8

Summary

The concepts learned from this book will help in understanding the basic operation of microprocessor-based systems. To further apply this information, study of actual microprocessor systems is encouraged along with hands-on experimentation with real components and circuits.

